Backup and hot standby for the LSS

6 Jan 2006
David Barrett-Lennard

Extension to the API

The API of the LSS will be extended in order to support level 0/1 backup as well as hot-standby.

The following functions will be added to ILogStructuredStore

```
// Delta Sequence Number
typedef int DSN;

interface IAsyncGetDeltaCallback
{
    virtual void OnFinished(DSN nextDsn) = 0;
};

interface ILogStructuredStore
{
    virtual bool AsyncGetDelta(
        DSN baseDsn, IOutputStream& os, IAsyncGetDeltaCallback& cb) = 0;

    virtual void DropBaseDSN(DSN baseDsn) = 0;

    virtual void ApplyDelta(IInputStream& is) = 0;

    < etc>
};
```

A Delta Sequence Number (DSN) begins at 0. Each DSN is associated with an *lss-delta*. An lss-delta is a byte stream that represents the serialised state of a set of objects to be applied as a delta to an LSS database. The lss-delta for DSN 0 is assumed to be applied to an empty LSS.

The lss-delta byte stream begins with a 32 bit count of the number of objects. Then for each object it writes the 64 bit OID, the 32 bit size of the serial element stream (-1 to indicate that the serial element is to be deleted), followed by the actual serial element stream (if any).

A call to `AsyncGetDelta()` signals the LSS to asynchronously build an lss-delta that can be used to bring a remote LSS (assumed to be in the state corresponding to baseDsn) up to date. If baseDsn = -1 then a complete snapshot of the LSS will be retrieved (and this can be compared to a level 0 backup). The delta is retrieved by a background worker thread, in such a way that it doesn't interfere with the ongoing operation of the LSS. ie the LSS will continue to support transactions, writing to disk, check pointing and cleaning. Note therefore that it is important that a consistent snapshot of the database be collected despite ongoing changes to objects.

A client calls `DropBaseDSN(baseDsn)` in order to tell the LSS that it no longer needs to support calls to `AsyncGetDelta()` up to and including the given baseDsn. This allows the LSS to purge some information from the store.

A client calls `ApplyDelta(is)` in order to apply an lss-delta (read from the given input stream) to the LSS. This synchronously writes/deletes all the serial elements in the lss-delta as a single transaction.

Application for hot-standby
An in-process layer on top of the LSS can listen on a port in order to accept connections from remote machines that want a hot-standby.

It is assumed that when an LSS is created, a GUID is generated that identifies the store. This helps ensure that a hot-standby client doesn't receive lss-deltas from an incompatible LSS.

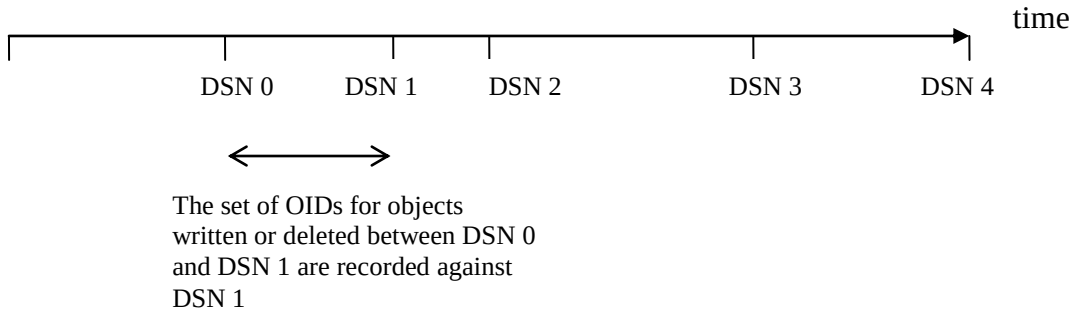
When a connection is accepted, the server sends the GUID to the client, allowing the client to validate against its stored GUID. The client then sends back its baseDSN, in effect telling the server where it is up to. The server can now send lss-deltas to the client in order to bring it up to date and keep it up to date. The client simply applies these deltas to its local LSS. Note that apart from this, no transactions on the client LSS can be performed.

Application for backup
An in-process layer on top of the LSS can listen on a port in order to accept connections from remote machines that want to perform level 0/1 backup of the LSS.

The approach is similar to the hot-standby, except that the deltas should be stored as files, using the DSN in the filename.

An exe could be written that builds an LSS from a given set of such files.

LSS Implementation notes



The LSS persists a mapping from DSN to a `std::set<OID>`. The set of OIDs are the objects that have been written or deleted since the last delta. Against DSN 0 we record the set of OIDs since the LSS was first created. Against DSN 1 we record the set of OIDs between the times that the deltas for DSN 0 and DSN 1 were generated, and so on.

In the above diagram we assume that DSN 4 is the next delta to be generated. Against this we are accumulating the *current set* of OIDs - ie since DSN 3.

Consider that the LSS receives a call to `AsyncGetDelta()`. Given a (valid) baseDsn, we can take the union of all the sets of OIDs for all DSN's - up to and including the

current set of OIDs for the next delta to be generated. This gives us the complete set of OIDs for which we need to generate the delta that will bring the remote LSS up to date (ie up to the "head").

The problem is to correctly generate a snapshot despite ongoing operation of the LSS. Consider that when `AsyncGetDelta()` is called we "briefly" get a lock on the RPM, and use this as an opportunity to find out and record the current locations of all the serial elements in the set<OID> to be processed. These locations are recorded as (segid, offset) pairs. If there is no entry in the RPM for a given OID we know we will indicate that the OID is to be deleted when the worker thread generates the lss-delta.

As long as relevant segments are not recycled, the worker thread will be able to correctly send a consistent snapshot of all the objects. At the time `AsyncGetDelta()` was called it was impossible for any of the segments to be either in the FSS or the delta-FSS because the segment utilisation must be non-zero. However, with continued operation of the LSS some segment utilisations may fall to zero causing the segids to be added to the delta-FSS. When a check point is performed we must prevent the delta-FSS from being added to the FSS (as is done normally). This should be as simple as setting a "protected mode" on the LSS while the worker thread gathers the snapshot.

Storing the set of OIDs

Consider that the set of OIDs is recorded in one or more segments, in a manner similar to the LargeSUT. ie segments a hijacked for this exclusive purpose as required. The segments form a chain. Over time new segments are added to the front and old segments are freed from the back.